Lab Code [10 points]
Filename: grader_test.sv
AndrewID: tbeasley

```
`default_nettype none

module Grader_test; // Test Grader module and Grader_FMS
  logic [11:0] Guess, Guess_pos, masterPattern;
  logic Grade_it_L, Gclr, Gload, CLOCK_50, reset_L;
  logic AeqB1, AeqB2, AeqB3, AeqB4;
  logic [3:0] Znarly, Zood, sum1, sum2, sum3, sum4, sum5, sum6, sum7;
  logic [2:0] Tshape, Cshape, Oshape, Dshape, Ishape, Zshape;
  logic [3:0] T_count, C_count, O_count, D_count, I_count, Z_count;
  logic [2:0] partial_Zoods;

  Grader g1(.*);
  Grader_FSM f1(.*);

  initial begin
    CLOCK_50 = 0;
    reset_L <= 0;
    forever #5 CLOCK_50 = ~CLOCK_50;
  end

  initial begin
    $monitor("masterPattern = %b | Guess = %b | %d Znarly(s) | %d Zood(s)",
              masterPattern, Guess, Znarly, Zood);

  // Moore Machine, will output correct value on next clock edge

  /* T = 001
     C = 010
     O = 011
     D = 100
     I = 101
     Z = 110 */

    @(posedge CLOCK_50); // INIT, no value
    reset_L <= 1; // ignore reset
    Grade_it_L <=1;
    @(posedge CLOCK_50); // INIT, no value


    Grade_it_L <= 0;
    @(posedge CLOCK_50); // SAVE, no value
    @(posedge CLOCK_50); // SAVE, no value

    Grade_it_L <= 1;
    @(posedge CLOCK_50); // HOLD, no value

    $display("//masterPattern 1//");

    masterPattern <= 12'b101_110_100_001; // IZDT
    Guess <= 12'b001_001_010_010; // TTCC
    @(posedge CLOCK_50); // HOLD, 0 Znarlys, 1 Zood

    Guess <= 12'b011_011_100_100; // OODD
    @(posedge CLOCK_50); // HOLD, 1 Znarly, 0 Zoods

    Guess <= 12'b101_101_010_010; // IICC
    @(posedge CLOCK_50); // HOLD, 1 Znarly, 0 Zoods

    Guess <= 12'b101_011_001_110; // IOTZ
    @(posedge CLOCK_50); // HOLD, 1 Znarly, 2 Zoods

    Guess <= 12'b001_101_110_100; // TIZD
    @(posedge CLOCK_50); // HOLD, 0 Znarlys, 4 Zoods

    Guess <= 12'b101_110_100_001; // IZDT
    @(posedge CLOCK_50); // HOLD, 4 Znarlys, 0 Zoods

    Grade_it_L <= 0;
    @(posedge CLOCK_50); // SAVE, clear Register Guess value
    @(posedge CLOCK_50);



    Grade_it_L <= 1;
    @(posedge CLOCK_50); // HOLD

    $display("//masterPattern 2//");

    masterPattern <= 12'b011_011_011_011; // OOOO
    Guess <= 12'b001_001_010_010; // TTCC
    @(posedge CLOCK_50); // HOLD, 0 Znarlys, 0 Zood

    Guess <= 12'b011_011_100_100; // OODD
    @(posedge CLOCK_50); // HOLD, 2 Znarly, 0 Zoods

    Guess <= 12'b101_101_010_010; // IICC
    @(posedge CLOCK_50); // HOLD, 0 Znarly, 0 Zoods

    Guess <= 12'b101_011_001_110; // IOTZ
    @(posedge CLOCK_50); // HOLD, 1 Znarly, 0 Zoods

    Guess <= 12'b001_101_110_100; // TIZD
    @(posedge CLOCK_50); // HOLD, 0 Znarlys, 0 Zoods

    Guess <= 12'b011_011_011_011; // 0000
    @(posedge CLOCK_50); // HOLD, 4 Znarlys, 0 Zoods

    Grade_it_L <= 0;
    @(posedge CLOCK_50); // SAVE, clear Register Guess value
    @(posedge CLOCK_50);

    #5 $finish;
  end

endmodule: Grader_test
```

Lab Code [10 points]
Filename: library.sv
AndrewID: tbeasley

```
`default_nettype none

module Decoder
  #(parameter WIDTH = 8)
  (input logic [$clog2(WIDTH) - 1:0] I,
  input logic en,
  output logic [WIDTH-1:0] D);

  assign D = (en) ? (2**I) : '0;

endmodule: Decoder

module BarrelShifter
  (input logic [15:0] V,
  input logic [3:0] by,
  output logic [15:0] S);

  assign S = V << by;

endmodule: BarrelShifter


module Multiplexer
  #(parameter WIDTH = 8)
  (input logic [WIDTH-1:0] I,
  input logic [$clog2(WIDTH) - 1:0] S,
  output logic Y);

  assign Y = I[S];

endmodule: Multiplexer


module Mux2to1
  #(parameter WIDTH = 7)
  (input logic [WIDTH-1:0] I0,
  input logic [WIDTH-1:0] I1,
  input logic S,
  output logic [WIDTH-1:0] Y);

  assign Y = (S === 1'b1) ? I1 : I0;


endmodule: Mux2to1


module MagComp
  #(parameter WIDTH = 8)
  (input logic [WIDTH-1:0] A,
  input logic [WIDTH-1:0] B,
  output logic AltB,
  output logic AeqB,
  output logic AgtB);

  assign AltB = (A < B);
  assign AeqB = (A === B);
  assign AgtB = (A > B);

endmodule: MagComp


module Comparator
  #(parameter WIDTH = 4)
  (input logic [WIDTH-1:0] A,
  input logic [WIDTH-1:0] B,
  output logic AeqB);

  assign AeqB = (A === B);

```

```

endmodule: Comparator


module Adder
  #(parameter WIDTH = 8)
  (input logic cin,
  input logic [WIDTH-1:0] A, B,
  output logic cout,
  output logic [WIDTH-1:0] sum);

  assign {cout, sum} = A + B + cin;

endmodule: Adder

module Subtracter
  #(parameter WIDTH = 8)
  (input logic bin,
  input logic [WIDTH-1:0] A, B,
  output logic bout,
  output logic [WIDTH-1:0] diff);

  assign {bout, diff} = A - B - bin;


endmodule: Subtracter


module DFlipFlop
  (input logic preset_L, D, clock, reset_L,
  output logic Q);

  always_ff @(posedge clock, negedge reset_L, negedge preset_L)
    if (~reset_L)
      Q <= 0;
    else if (~preset_L)
      Q <= 1;
    else
      Q <= D;

endmodule: DFlipFlop


module Register
  #(parameter WIDTH = 8)
  (input logic en, clear, clock,
  input logic [WIDTH - 1:0] D,
  output logic [WIDTH - 1:0] Q);

  always_ff @(posedge clock)
    if (en)
      Q <= D;
    else if (clear)
      Q <= '0;

endmodule: Register


module Counter
  #(parameter WIDTH = 8)
  (input logic en, clear, load, up, clock,
  input logic [WIDTH-1:0] D,
  output logic [WIDTH-1:0] Q);

   always_ff @(posedge clock)
    if (clear)
      Q <= '0;
    else if (load)
      Q <= D;
    else if (up && en)
      Q <= Q + 1'd1;
    else if (~up && en)
      Q <= Q - 1'd1;

endmodule: Counter


module Synchronizer
  (input logic async, clock,
  output logic sync);

    logic buffer;

    always_ff @(posedge clock) begin
      buffer <= async;
      sync <= buffer;
    end

endmodule: Synchronizer


module ShiftRegisterSIPO
  #(parameter WIDTH = 8)
  (input logic en, left, serial, clock,
  output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (left && en)
      Q <= {Q[WIDTH-2:0], serial};
    else if (~left && en)
      Q <= {serial, Q[WIDTH-1:1]};

endmodule: ShiftRegisterSIPO


module ShiftRegisterPIPO
  #(parameter WIDTH = 8)
  (input logic en, left, load, clock,
  input logic [WIDTH-1:0] D,
  output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (load)
      Q <= D;
    else if (left && en && ~load)
      Q <= Q << 1;
    else if (~left && en && ~load)
      Q <= Q >> 1;

endmodule: ShiftRegisterPIPO


module BarrelShiftRegister
  #(parameter WIDTH = 8)
  (input logic en, load, clock,
  input logic [1:0] by,
  input logic [WIDTH-1:0] D,
  output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (load)
      Q <= D;
    else if (en)
      Q <= Q << by;

endmodule: BarrelShiftRegister

module BusDriver
  #(parameter WIDTH = 8)
  (input logic en,
  input logic [WIDTH-1:0] data,
  output logic [WIDTH-1:0] buff,
  inout tri [WIDTH-1:0] bus);

  assign bus = (en) ? data : 'bz;
  assign buff = bus;

endmodule: BusDriver


module Memory
  #(parameter DW = 16,
              W  = 256,
              AW = $clog2(W))
  (input logic re, we, clock,
  input logic [AW-1:0] addr,
  inout tri [DW-1:0] data);

  logic [DW-1:0] M[W];
  logic [DW-1:0] rData;

  assign data = (re) ? rData: 'bz;

  always_ff @(posedge clock)
    if (we)
      M[addr] <= data;

  always_comb
    rData = M[addr];

endmodule: Memory
```

Lab Code [10 points]
Filename: mastermindVGA.sv
AndrewID: tbeasley

```
/** FILE
 *  mastermindVGA.sv
 *
 *  BRIEF
 *  Module that acts an an interface between the Mastermind game and
 *  the VGA output.
 *
 *  The game field will look like a standard Mastermind playing field,
 *  with a small number in the lower right indicating the number of games
 *  available.
 *
 *  Zorgian terminology:
 *  Znarly = correct shape, correct spot
 *  Zood   = correct shape, wrong spot
 *
 *  AUTHOR
 *  Anita Zhang (anitazha)
 */

/****** File-wide Colors ***********/
typedef enum logic [23:0] {
    RED      = {8'hFF, 8'h00, 8'h00},
    GREEN    = {8'h00, 8'hFF, 8'h00},
    BLUE     = {8'h00, 8'h00, 8'hFF},
    CYAN     = {8'h00, 8'hFF, 8'hCC},
    PURPLE   = {8'h99, 8'h00, 8'hFF},
    YELLOW   = {8'hFF, 8'hFF, 8'h00},
    BLACK    = {8'h00, 8'h00, 8'h00},
    WHITE    = {8'hFF, 8'hFF, 8'hFF}
} color_t;

/****** File-wide Shapes ***********/
typedef enum logic [2:0] {
    LEFTTOP  = 3'b001,  // blue
    WALL     = 3'b010,  // red
    RIGHTTOP = 3'b011,  // cyan
    EQUAL    = 3'b100,  // purple
    RIGHTBOT = 3'b101,  // green
    LEFTBOT  = 3'b110   // yellow
} shape_t;

/** BRIEF
 *  Main module that handles user input and displays game data.
 */

module mastermindVGA (
    input  logic        CLOCK_50,
    // VGA display signals -- route directly to FPGA pins
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_BLANK_N, VGA_CLK, VGA_SYNC_N,
    output logic        VGA_VS, VGA_HS,
    // game information
    input  logic [3:0]  numGames,
    input  logic        loadNumGames,
    // Items for a particular round
    input  logic [3:0]  roundNumber,
    input  logic [11:0] guess,
    input  logic        loadGuess,
    input  logic [3:0]  znarly, zood,
    input  logic        loadZnarlyZood,
    input  logic        clearGame,
    // master patterns
    input  logic [11:0] masterPattern,
    input  logic        displayMasterPattern,
    // other
    input  logic        reset
    );

    /****************************************
```

```
     *       Internal Signals
     ****************************************/

    // game data
    logic [7:0][3:0][2:0] memGuess;
    logic [7:0][3:0]      memZnarly;
    logic [7:0][3:0]      memZood;
    logic [3:0]           memNumGames;
    logic [3:0][2:0]      master;
    // VGA data
    logic [9:0]           x, y;
    logic                 blank;
    // drawing data
    logic [2:0]           shapeSel;
    logic [3:0]           numValue;
    logic [1:0]           masterIdx, guessIdxX;
    logic [2:0]           guessIdxY;
    logic [3:0]           gIdxX, gIdxY;
    logic                 inGameZoneX, inGameZoneY;
    logic                 isNum, isCredit, isRound;
    logic                 isMaster, isZZ, isShape;
    color_t               zzColor, shapeColor;
    color_t               color;
    // loop counters
    integer               i, j;
    // other
    logic                 clk;

    // game playing field
    localparam X0 = 10'd169;
    localparam X1 = 10'd481;
    localparam Y0 = 10'd10;
    localparam Y1 = 10'd468;
    localparam GSIDE = 10'd52;    // grid width
    // specific playing field coordinates
    localparam ZZ_X = 10'd429;    // znarly/zood X position
    localparam MASTER_Y = 10'd426; // master pattern Y position

    // renamed signals
    assign clk = CLOCK_50;
    assign master = masterPattern;

    /****************************************
     *       VGA data
     ****************************************/

    vga vgaCounter (
            .row        (y),
            .col        (x),
            .HS         (VGA_HS),
            .VS         (VGA_VS),
            .*);

    assign VGA_BLANK_N           = ~blank;
    assign VGA_CLK               = CLOCK_50;
    assign VGA_SYNC_N            = 1'b0;
    assign {VGA_R, VGA_G, VGA_B} = color;

    /****************************************
     *          Store Game Info
     ****************************************/

    registerAZ #(4) numGamesReg (
            .Q      (memNumGames),
            .D      (numGames),
            .clr    (clearGame),
            .en     (loadNumGames),
            .*);

    // have guess separate so the switch flipping is displayed
    always_ff @(posedge clk, posedge reset) begin
```

```
        if (reset)
            memGuess <= 96'b0;
        else if (clearGame)
            memGuess <= 96'b0;
        else if (loadGuess)
            memGuess[roundNumber] <= guess;
    end

    // only store znarly and zood when ready
    always_ff @(posedge clk, posedge reset) begin
        if (reset) begin
            memZnarly <= 32'b0;
            memZood <= 32'b0;
        end
        else if (clearGame) begin
            memZnarly <= 32'b0;
            memZood <= 32'b0;
        end
        else if (loadZnarlyZood) begin
            memZood[roundNumber]   <= zood;
            memZnarly[roundNumber] <= znarly;
        end
    end

    /****************************************
     *          Color/Boundary Assignments
     ****************************************/

    range_check gameFieldX (
            .val        (x),
            .low        (X0),
            .high       (X1),
            .is_between (inGameZoneX)
            );

    range_check gameFieldY (
            .val        (y),
            .low        (Y0),
            .high       (Y1),
            .is_between (inGameZoneY)
            );

    always_comb begin
        color = BLACK;
        if (inGameZoneX & inGameZoneY) begin
            // round number
            if (isRound & isNum)
                color = WHITE;
            // credits
            else if (isCredit & isNum)
                color = CYAN;
            // znarly/zood
            else if (isZZ)
                color = zzColor;
            // master shape
            else if (isMaster & displayMasterPattern)
                color = shapeColor;
            // guess shape
            else if (isShape)
                color = shapeColor;
        end
    end

    /****************************************
     *       Draw Things
     ****************************************/

    // grid index (of the playing field) -- "grid" is 6 x 9
    assign gIdxX     = (inGameZoneX ? ((x - X0) / GSIDE) : 4'b1111);
    assign gIdxY     = (inGameZoneY ? ((y - Y0) / GSIDE) : 4'b1111);

    // define "zones" for each shape type
    assign isRound   = ((gIdxX == 4'd0) & (gIdxY < 4'd8));
    assign isZZ      = ((gIdxX == 4'd5) & (gIdxY < 4'd8));
    assign isCredit  = ((gIdxX == 4'd5) & (gIdxY == 4'd8));
    assign isMaster  = ((gIdxX > 4'd0) & (gIdxX < 4'd5) & (gIdxY == 4'd8));
    assign isShape   = ((gIdxX > 4'd0) & (gIdxX < 4'd5) & (gIdxY < 4'd8));

    // indices/signals for drawing the shapes
    assign numValue  = (isCredit ? memNumGames : gIdxY);
    assign shapeSel  = (isMaster ? master[masterIdx] :
                        memGuess[guessIdxY][guessIdxX]);
    assign masterIdx = (isMaster ? ~(gIdxX - 1'b1) : 2'b0);
    assign guessIdxX = (isShape  ? ~(gIdxX - 1'b1) : 2'b0);
    assign guessIdxY = (isShape  ? gIdxY : 4'b0);

    // draw the round numbers on the side, or the credit value
    drawNumber numDrawer (
            .inNum  (isNum),
            .x      (x),
            .y      (y),
            .posX   (X0 + (gIdxX * GSIDE)),
            .posY   (Y0 + (gIdxY * GSIDE)),
            .value  (numValue)
            );

    // draw shapes for the guess field
    drawShape shapeDrawer (
            .color  (shapeColor),
            .x      (x),
            .y      (y),
            .posX   (X0 + (gIdxX * GSIDE)),
            .posY   (Y0 + (gIdxY * GSIDE)),
            .shape  (shapeSel)
            );

    // draw Znarly/Zood results
    drawZnarlyZood zzDrawer (
            .color  (zzColor),
            .znarly (memZnarly[gIdxY]),
            .zood   (memZood[gIdxY]),
            .x      (x),
            .y      (y),
            .posX   (ZZ_X),
            .posY   (Y0 + (gIdxY * GSIDE))
            );

endmodule: mastermindVGA

/*****************************************************************
 *
 *                     Drawing modules
 *
 *****************************************************************/

/** BRIEF
 *  Given the position of a 42x42 px box (the upper left coordinate),
 *  draw a number specified by "value". One of the inputs will be the
 *  current (x, y) coordinate being processed, and a bit will be
 *  output according to whether that pixel is in the number's zone
 */
module drawNumber
    #(parameter LINEWIDTH = 10'd4, PADDING = 10'd10, SIDE = 10'd42) (
    output logic        inNum,
    input  logic [9:0]  x, y,
    input  logic [9:0]  posX, posY,
    input  logic [2:0]  value
    );

    // internal signals
    logic   [6:0] isSegX, isSegY, isSeg;

    /****************************************
     *          Output logic
     ****************************************/

    assign isSeg = (isSegX & isSegY);

    always_comb begin
        inNum = 1'b0;

        case (value)
            3'd0: begin
                if (isSeg[5:0])
                    inNum = 1'b1;
            end
            3'd1: begin
                if (isSeg[2:1])
                    inNum = 1'b1;
            end
            3'd2: begin
                if (isSeg[0] | isSeg[1] | isSeg[6] | isSeg[4] | isSeg[3])
                    inNum = 1'b1;
            end
            3'd3: begin
                if (isSeg[3:0] || isSeg[6])
                    inNum = 1'b1;
            end
            3'd4: begin
                if (isSeg[6:5] || isSeg[2:1])
                    inNum = 1'b1;
            end
            3'd5: begin
                if (isSeg[6:5] || isSeg[3:2] || isSeg[0])
                    inNum = 1'b1;
            end
            3'd6: begin
                if (isSeg[6:2] || isSeg[0])
                    inNum = 1'b1;
            end
            3'd7: begin
                if (isSeg[2:0])
                    inNum = 1'b1;
            end
        endcase
    end

    /****************************************
     *          Segment Boundary Check
     ****************************************/

    // top segment
    offset_check #(10) segCheckX0 (
            .val        (x),
            .low        (posX + PADDING),
            .delta      (SIDE - (2*PADDING)),
            .is_between (isSegX[0]));

    offset_check #(10) segCheckY0 (
            .val        (y),
            .low        (posY + PADDING),
            .delta      (LINEWIDTH),
            .is_between (isSegY[0]));

    // top right segment
    offset_check #(10) segCheckX1 (
            .val        (x),
            .low        (posX + (SIDE - PADDING) - LINEWIDTH),
            .delta      (LINEWIDTH),
            .is_between (isSegX[1]));

    offset_check #(10) segCheckY1 (
            .val        (y),
            .low        (posY + PADDING),
            .delta      ((SIDE - (PADDING*2))/2),
            .is_between (isSegY[1]));

    // bottom right segment
    offset_check #(10) segCheckX2 (
            .val        (x),
            .low        (posX + (SIDE - PADDING) - LINEWIDTH),
            .delta      (LINEWIDTH),
            .is_between (isSegX[2]));

    offset_check #(10) segCheckY2 (
            .val        (y),
            .low        (posY + PADDING + ((SIDE - (2*PADDING))/2)),
            .delta      ((SIDE - (PADDING*2))/2),
            .is_between (isSegY[2]));

    // bottom segment
    offset_check #(10) segCheckX3 (
            .val        (x),
            .low        (posX + PADDING),
            .delta      (SIDE - (2*PADDING)),
            .is_between (isSegX[3]));

    offset_check #(10) segCheckY3 (
            .val        (y),
            .low        (posY + (SIDE - PADDING) - LINEWIDTH),
            .delta      (LINEWIDTH),
            .is_between (isSegY[3]));

    // bottom left segment
    offset_check #(10) segCheckX4 (
            .val        (x),
            .low        (posX + PADDING),
            .delta      (LINEWIDTH),
            .is_between (isSegX[4]));

    offset_check #(10) segCheckY4 (
            .val        (y),
            .low        (posY + PADDING + ((SIDE - (2*PADDING))/2)),
            .delta      ((SIDE - (PADDING*2))/2),
            .is_between (isSegY[4]));

    // top left segment
    offset_check #(10) segCheckX5 (
            .val        (x),
            .low        (posX + PADDING),
            .delta      (LINEWIDTH),
            .is_between (isSegX[5]));

    offset_check #(10) segCheckY5 (
            .val        (y),
            .low        (posY + PADDING),
            .delta      ((SIDE - (PADDING*2))/2),
            .is_between (isSegY[5]));

    // middle segment
    offset_check #(10) segCheckX6 (
            .val        (x),
            .low        (posX + PADDING),
            .delta      (SIDE - (2*PADDING)),
            .is_between (isSegX[6]));

    offset_check #(10) segCheckY6 (
            .val        (y),
            .low        (posY + (SIDE/2) - LINEWIDTH/2),
            .delta      (LINEWIDTH),
            .is_between (isSegY[6]));

endmodule: drawNumber

```

```
/** BRIEF
 *  Given the position of a 42x42 px box (the upper left coordinate),
 *  draw the znarly/zood results. One of the inputs will be the
 *  current (x, y) coordinate being processed, and a color will be
 *  output according to whether that pixel is in the shape's zone.
 *
 *  Znarly is red; zood white (because I saw it on the internet)
 */
module drawZnarlyZood
    #(parameter WIDTH = 10'd26, PADDING = 10'd16) (
    output color_t        color,
    input  logic   [3:0]  znarly, zood,
    input  logic   [9:0]  x, y,
    input  logic   [9:0]  posX, posY
    );

    /****************************************
     *       Internal Signals
     ****************************************/

    logic   [3:0] inBoxX, inBoxY;

    /****************************************
     *       Boundary Checks
     ****************************************/

    // create boundaries for 4 tiny squares
    // top left
    offset_check #(10) squareCheckX0 (
            .val        (x),
            .low        (posX),
            .delta      (PADDING),
            .is_between (inBoxX[0]));

    offset_check #(10) squareCheckY0 (
            .val        (y),
            .low        (posY),
            .delta      (PADDING),
            .is_between (inBoxY[0]));

    // top right
    offset_check #(10) squareCheckX1 (
            .val        (x),
            .low        (posX + WIDTH),
            .delta      (PADDING),
            .is_between (inBoxX[1]));

    offset_check #(10) squareCheckY1 (
            .val        (y),
            .low        (posY),
            .delta      (PADDING),
            .is_between (inBoxY[1]));

    // bottom left
    offset_check #(10) squareCheckX2 (
            .val        (x),
            .low        (posX),
            .delta      (PADDING),
            .is_between (inBoxX[2]));

    offset_check #(10) squareCheckY2 (
            .val        (y),
            .low        (posY + WIDTH),
            .delta      (PADDING),
            .is_between (inBoxY[2]));

    // bottom right
    offset_check #(10) squareCheckX3 (
            .val        (x),
            .low        (posX + WIDTH),
            .delta      (PADDING),
            .is_between (inBoxX[3]));

    offset_check #(10) squareCheckY3 (
            .val        (y),
            .low        (posY + WIDTH),
            .delta      (PADDING),
            .is_between (inBoxY[3]));

    /****************************************
     *       Color Assignments
     ****************************************/

    always_comb begin
        color = BLACK;

        if (inBoxX[0] & inBoxY[0]) begin
            if (znarly > 3'd0)
                color = RED;
            else if (zood > 3'd3)
                color = WHITE;
        end
        else if (inBoxX[1] & inBoxY[1]) begin
            if (znarly > 3'd1)
                color = RED;
            else if (zood > 3'd2)
                color = WHITE;
        end
        else if (inBoxX[2] & inBoxY[2]) begin
            if (znarly > 3'd2)
                color = RED;
            else if (zood > 3'd1)
                color = WHITE;
        end
        else if (inBoxX[3] & inBoxY[3]) begin
            if (znarly > 3'd3)
                color = RED;
            else if (zood > 3'd0)
                color = WHITE;
        end
    end

endmodule: drawZnarlyZood

/** BRIEF
 *  Given the position of a 42x42 px box (the upper left coordinate),
 *  draw a shape specified by "shape". One of the inputs will be the
 *  current (x, y) coordinate being processed, and a color will be
 *  output according to whether that pixel is in the shape's zone.
 */
module drawShape
    #(parameter LINEWIDTH = 10'd15, SIDE = 10'd42) (
    output color_t        color,
    input  logic   [9:0]  x, y,
    input  logic   [9:0]  posX, posY,
    input  logic   [2:0]  shape
    );

    /****************************************
     *       Internal Signals
     ****************************************/

    color_t shapeColor;
    shape_t shapeType;
    logic   inBoxX, inBoxY;
    logic   inTopStick, inBottomStick, inLeftStick, inRightStick;

    /****************************************
     *       Boundary Checks
     ****************************************/

    // create square boundaries
    offset_check #(10) squareCheckX (
            .val        (x),
            .low        (posX),
            .delta      (SIDE),
            .is_between (inBoxX));

    offset_check #(10) squareCheckY (
            .val        (y),
            .low        (posY),
            .delta      (SIDE),
            .is_between (inBoxY));

    // create a top border inside the box
    offset_check #(10) topStick (
            .val        (y),
            .low        (posY),
            .delta      (LINEWIDTH),
            .is_between (inTopStick));

    // create a bottom border inside the box
    offset_check #(10) bottomStick (
            .val        (y),
            .low        (posY + SIDE - LINEWIDTH),
            .delta      (LINEWIDTH),
            .is_between (inBottomStick));

    // create a left border inside the box
    offset_check #(10) leftStick (
            .val        (x),
            .low        (posX),
            .delta      (LINEWIDTH),
            .is_between (inLeftStick));

    // create a right border inside the box
    offset_check #(10) rightStick (
            .val        (x),
            .low        (posX + SIDE - LINEWIDTH),
            .delta      (LINEWIDTH),
            .is_between (inRightStick));

    /****************************************
     *       Final Output logic
     ****************************************/

    assign color = shapeColor;
    assign shapeType = shape_t'(shape);

    always_comb begin
        shapeColor = BLACK;

        if (inBoxX & inBoxY) begin
            if ((shapeType == WALL) & (inLeftStick | inRightStick)) begin
                shapeColor = RED;
            end
            else if ((shapeType == LEFTTOP) & (inLeftStick | inTopStick)) begin
                shapeColor = BLUE;
            end
            else if ((shapeType == RIGHTTOP) & (inRightStick | inTopStick))
            begin
                shapeColor = CYAN;
            end
            else if ((shapeType == RIGHTBOT) & (inRightStick | inBottomStick))
            begin
                shapeColor = GREEN;
            end
            else if ((shapeType == LEFTBOT) & (inLeftStick | inBottomStick))
            begin
                shapeColor = YELLOW;
            end
            else if ((shapeType == EQUAL) & (inBottomStick | inTopStick)) begin
                shapeColor = PURPLE;
            end
        end
    end
endmodule: drawShape

/*****************************************************************
 *
 *                    VGA Magic
 *
 *****************************************************************/

/** BRIEF
 *  VGA module that outputs the current hsync and vsync values needed
 *  to display content. Does not handle the actual color content.
 *
 *  Requires the Library.sv modules to work. Supports 640 x 480 px.
 */
module vga (
    output logic [9:0] row, col,
    output logic       HS, VS, blank,
    input  logic       CLOCK_50, reset
    );

    logic [10:0] col_count;
    logic        col_clear, col_enable;
    logic [9:0]  row_count;
    logic        row_clear, row_enable;
    logic        h_blank, v_blank;

    // Row counter counts from 0 to 520
    //     count of   0 - 479 is display time (thus row_count is correct here)
    //     count of 480 - 489 is front porch
    //     count of 490 - 491 is VS=0 pulse width
    //     count of 492 - 520 is back porch

    simple_counter #(10) row_counter(
            .Q      (row_count),
            .en     (row_enable),
            .clr    (row_clear),
            .clk    (CLOCK_50),
            .reset  (reset)
            );

    assign row        = row_count;
    assign row_clear  = (row_count >= 10'd520);
    assign row_enable = (col_count == 11'd1599);
    assign VS         = (row_count < 10'd490) | (row_count > 10'd491);
    assign v_blank    = (row_count >= 10'd480);

    // Col counter counts from 0 to 1599
    //     count of    0 - 1279 is display time (col is div by 2)
    //     count of 1280 - 1311 is front porch
    //     count of 1312 - 1503 is HS=0 pulse width
    //     count of 1504 - 1599 is back porch

    simple_counter #(11) col_counter(
            .Q      (col_count),
            .en     (col_enable),
            .clr    (col_clear),
            .clk    (CLOCK_50),
            .reset  (reset)
            );

    assign col        = col_count[10:1];
    assign col_clear  = (col_count >= 11'd1599);
    assign col_enable = 1'b1;
    assign HS         = (col_count < 11'd1312) | (col_count > 11'd1503);
    assign h_blank    = col_count > 11'd1279;

    assign blank      = h_blank | v_blank;
endmodule: vga
```

```

/******************************************************************
 *
 *                    Library modules
 *
 ******************************************************************/

/** BRIEF
 *  Outputs whether a value lies between [low, high].
 */
module range_check
    #(parameter WIDTH = 4'd10) (
    input  logic [WIDTH-1:0] val, low, high,
    output logic             is_between
    );

    assign is_between = (val >= low) & (val <= high);

endmodule: range_check

/** BRIEF
 *  Outputs whether a value lies between [low, low + delta].
 */
module offset_check
    #(parameter WIDTH = 4'd10) (
    input  logic [WIDTH-1:0] val, low, delta,
    output logic             is_between
    );

    assign is_between = ((val >= low) & (val < (low+delta)));

endmodule: offset_check

/** BRIEF
 *  Simple up counter with synchronous clear and enable.
 *  Clear takes precedence over enable.
 */
module simple_counter
    #(parameter WIDTH = 4'd8) (
    output logic [WIDTH-1:0] Q,
    input  logic             clk, en, clr, reset
    );

    always_ff @(posedge clk, posedge reset)
        if (reset)
            Q <= 'b0;
        else if (clr)
            Q <= 'b0;
        else if (en)
            Q <= (Q + 1'b1);

endmodule: simple_counter

/** BRIEF
 *  A register with synchronous clear. Clear takes precedence.
 */
module registerAZ
    #(parameter WIDTH = 4'd8) (
    output logic [WIDTH-1:0] Q,
    input  logic [WIDTH-1:0] D,
    input  logic             clk, en, clr, reset
    );

    always_ff @(posedge clk, posedge reset) begin
        if (reset)
            Q <= 'b0;
        else if (clr)
            Q <= 'b0;
        else if (en)
            Q <= D;
    end
```

```
endmodule: registerAZ
```

Lab Code [10 points]
Filename: zorgGame.sv
AndrewID: tbeasley

```
`default_nettype none

module ChipInterface // Chip Interface for FPGA and VGA connections
  (input logic [17:0] SW,
  input logic [3:0] KEY,
  input logic CLOCK_50,
  output logic [7:0] LEDG,
  output logic [8:0] LEDR,
  output logic [6:0] HEX3, HEX2, HEX1, HEX0,
  output logic [7:0]  VGA_R, VGA_G, VGA_B,
  output logic        VGA_BLANK_N, VGA_CLK, VGA_SYNC_N,
  output logic        VGA_VS, VGA_HS);

  logic [3:0] znarlyOut, zoodOut, roundOut, gamesOut;
  logic [11:0] masterOut;
  logic GameWonOut, maxRoundOut, clearGameOut, done, loadValue, syncGradeIt;

  assign clearGameOut = (GameWonOut|maxRoundOut);
  assign LEDG[0] = GameWonOut;


  ZorgGame game(.CoinValue(SW[17:16]), .ShapeLocation(SW[4:3]),
                .CoinInserted(KEY[1]), .StartGame(KEY[2]), .GradeIt(KEY[3]),
                .LoadShapeNow(KEY[3]), .reset(KEY[0]), .clock(CLOCK_50),
                .Guess(SW[11:0]), .LoadShape(SW[2:0]),.Znarly(znarlyOut),
                .Zood(zoodOut), .RoundNumber(roundOut), .NumGames(gamesOut),
                .GameWon(GameWonOut), .maxRound(maxRoundOut),
                .masterPattern(masterOut), .doneMasterPattern(done),
                .syncGradeIt);


  mastermindVGA vgaDisplay(.CLOCK_50, .VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N,
                           .VGA_CLK, .VGA_SYNC_N, .VGA_VS, .VGA_HS,
                           .numGames(gamesOut), .loadNumGames(1'b1),
                           .roundNumber(roundOut), .guess(SW[11:0]),
                           .loadGuess(done), .znarly(znarlyOut),
                           .zood(zoodOut), .loadZnarlyZood(syncGradeIt),
                           .clearGame(clearGameOut), .masterPattern(masterOut),
                           .displayMasterPattern(SW[15]), .reset(~KEY[0]));


  SevenSegmentDisplay seg(.BCD0(gamesOut), .BCD1(roundOut), .BCD2(zoodOut),
                          .BCD3(znarlyOut), .blank(8'b1111_0000), .HEX0, .HEX1,
                          .HEX2, .HEX3);

endmodule: ChipInterface


// Expanded version of the abstract FSM implemented in Task 2
module myAbstractFSMExpanded (
  output logic [3:0] credit,
  output logic drop,
  input logic [1:0] CoinValue,
  input logic CoinInserted_L, clock, reset_L);

  enum logic [4:0] {NOCREDIT, HOLDC00, HOLDT00, HOLDP00,
                    CRED1DROP0, HOLDC10, HOLDT10, HOLDP10,
                    CRED2DROP0, HOLDC20, HOLDT20, HOLDP20,
                    CRED3DROP0, HOLDC30, HOLDT30, HOLDP30,
                    CRED0DROP1, HOLDC01, HOLDT01, HOLDP01,
                    CRED1DROP1, HOLDC11, HOLDT11, HOLDP11,
                    CRED2DROP1, HOLDC21, HOLDT21, HOLDP21,
                    CRED3DROP1, HOLDC31, HOLDT31, HOLDP31} currState, nextState;

//Next State Logic
always_comb begin
  case (currState)
    NOCREDIT: begin
        if (CoinValue == 2'b01 & ~CoinInserted_L)
```

```
          nextState = HOLDC00;
        else if (CoinValue == 2'b10 & ~CoinInserted_L)
          nextState = HOLDT00;
        else if (CoinValue == 2'b11 & ~CoinInserted_L)
          nextState = HOLDP00;
        else
          nextState = NOCREDIT;
    end
    HOLDC00: begin
        if (~CoinInserted_L)
          nextState = HOLDC00;
        else
          nextState = CRED1DROP0;
    end
    HOLDT00: begin
        if (~CoinInserted_L)
          nextState = HOLDT00;
        else
          nextState = CRED3DROP0;
    end
    HOLDP00: begin
        if (~CoinInserted_L)
          nextState = HOLDP00;
        else
          nextState = CRED1DROP1;
    end

    CRED1DROP0: begin
        if (CoinValue == 2'b01 & ~CoinInserted_L)
          nextState = HOLDC10;
        else if (CoinValue == 2'b10 & ~CoinInserted_L)
          nextState = HOLDT10;
        else if (CoinValue == 2'b11 & ~CoinInserted_L)
          nextState = HOLDP10;
        else
          nextState = CRED1DROP0;
    end
    HOLDC10: begin
        if (~CoinInserted_L)
          nextState = HOLDC10;
        else
          nextState = CRED2DROP0;
    end
    HOLDT10: begin
        if (~CoinInserted_L)
          nextState = HOLDT10;
        else
          nextState = CRED0DROP1;
    end
    HOLDP10: begin
        if (~CoinInserted_L)
          nextState = HOLDP10;
        else
          nextState = CRED2DROP1;
    end

    CRED2DROP0: begin
        if (CoinValue == 2'b01 & ~CoinInserted_L)
          nextState = HOLDC20;
        else if (CoinValue == 2'b10 & ~CoinInserted_L)
          nextState = HOLDT20;
        else if (CoinValue == 2'b11 & ~CoinInserted_L)
          nextState = HOLDP20;
        else
          nextState = CRED2DROP0;
    end
    HOLDC20: begin
        if (~CoinInserted_L)
          nextState = HOLDC20;
        else
          nextState = CRED3DROP0;
    end
    HOLDT20: begin
        if (~CoinInserted_L)
          nextState = HOLDT20;
        else
          nextState = CRED1DROP1;
    end
    HOLDP20: begin
        if (~CoinInserted_L)
          nextState = HOLDP20;
        else
          nextState = CRED3DROP1;
    end

    CRED3DROP0: begin
        if (CoinValue == 2'b01 & ~CoinInserted_L)
          nextState = HOLDC30;
        else if (CoinValue == 2'b10 & ~CoinInserted_L)
          nextState = HOLDT30;
        else if (CoinValue == 2'b11 & ~CoinInserted_L)
          nextState = HOLDP30;
        else
          nextState = CRED3DROP0;
    end
    HOLDC30: begin
        if (~CoinInserted_L)
          nextState = HOLDC30;
        else
          nextState = CRED0DROP1;
    end
    HOLDT30: begin
        if (~CoinInserted_L)
          nextState = HOLDT30;
        else
          nextState = CRED2DROP1;
    end
    HOLDP30: begin
        if (~CoinInserted_L)
          nextState = HOLDP30;
        else
          nextState = CRED0DROP1;
    end

    CRED0DROP1: begin
        if (CoinValue == 2'b01 & ~CoinInserted_L)
          nextState = HOLDC01;
        else if (CoinValue == 2'b10 & ~CoinInserted_L)
          nextState = HOLDT01;
        else if (CoinValue == 2'b11 & ~CoinInserted_L)
          nextState = HOLDP01;
        else
          nextState = NOCREDIT;
    end
    HOLDC01: begin
        if (~CoinInserted_L)
          nextState = HOLDC01;
        else
          nextState = CRED1DROP0;
    end
    HOLDT01: begin
        if (~CoinInserted_L)
          nextState = HOLDT01;
        else
          nextState = CRED3DROP0;
    end
    HOLDP01: begin
        if (~CoinInserted_L)
          nextState = HOLDP01;
        else
          nextState = CRED1DROP1;
    end
    CRED1DROP1: begin
        if (CoinValue == 2'b01 & ~CoinInserted_L)
          nextState = HOLDC11;
        else if (CoinValue == 2'b10 & ~CoinInserted_L)
          nextState = HOLDT11;
        else if (CoinValue == 2'b11 & ~CoinInserted_L)
          nextState = HOLDP11;
        else
          nextState = CRED1DROP0;
    end
    HOLDC11: begin
        if (~CoinInserted_L)
          nextState = HOLDC11;
        else
          nextState = CRED2DROP0;
    end
    HOLDT11: begin
        if (~CoinInserted_L)
          nextState = HOLDT11;
        else
          nextState = CRED0DROP1;
    end
    HOLDP11: begin
        if (~CoinInserted_L)
          nextState = HOLDP11;
        else
          nextState = CRED2DROP1;
    end

    CRED2DROP1: begin
        if (CoinValue == 2'b01 & ~CoinInserted_L)
          nextState = HOLDC21;
        else if (CoinValue == 2'b10 & ~CoinInserted_L)
          nextState = HOLDT21;
        else if (CoinValue == 2'b11 & ~CoinInserted_L)
          nextState = HOLDP21;
        else
          nextState = CRED2DROP0;
    end
    HOLDC21: begin
        if (~CoinInserted_L)
          nextState = HOLDC21;
        else
          nextState = CRED3DROP0;
    end
    HOLDT21: begin
        if (~CoinInserted_L)
          nextState = HOLDT21;
        else
          nextState = CRED1DROP1;
    end
    HOLDP21: begin
        if (~CoinInserted_L)
          nextState = HOLDP21;
        else
          nextState = CRED3DROP1;
    end

    CRED3DROP1: begin
        if (CoinValue == 2'b01 & ~CoinInserted_L)
          nextState = HOLDC31;
        else if (CoinValue == 2'b10 & ~CoinInserted_L)
          nextState = HOLDT31;
        else if (CoinValue == 2'b11 & ~CoinInserted_L)
          nextState = HOLDP31;
        else
          nextState = CRED3DROP0;
    end
    HOLDC31: begin
        if (~CoinInserted_L)
```

```
     nextState = HOLDC31;
    else
     nextState = CRED0DROP1;
  end
  HOLDT31: begin
    if (~CoinInserted_L)
     nextState = HOLDT31;
    else
     nextState = CRED2DROP1;
  end
  HOLDP31: begin
    if (~CoinInserted_L)
     nextState = HOLDP31;
    else
     nextState = CRED0DROP1;
  end

  default: begin
    nextState = NOCREDIT;
  end
 endcase
end

//Output logic
always_comb begin
 credit = 4'b0000; drop = 1'b0;
 unique case (currState)
  NOCREDIT: begin
   drop = 1'b0;
   credit = 4'b0000;
  end
  HOLDC00: begin
   drop = 1'b0;
   credit = 4'b0000;
  end
  HOLDT00: begin
   drop = 1'b0;
   credit = 4'b0000;
  end
  HOLDP00: begin
   drop = 1'b0;
   credit = 4'b0000;
  end

  CRED1DROP0: begin
   drop = 1'b0;
   credit = 4'b0001;
  end
  HOLDC10: begin
   drop = 1'b0;
   credit = 4'b0001;
  end
  HOLDT10: begin
   drop = 1'b0;
   credit = 4'b0001;
  end
  HOLDP10: begin
   drop = 1'b0;
   credit = 4'b0001;
  end

  CRED2DROP0: begin
   drop = 1'b0;
   credit = 4'b0010;
  end
  HOLDC20: begin
   drop = 1'b0;
   credit = 4'b0010;
  end
  HOLDT20: begin
   drop = 1'b0;
```

```
      credit = 4'b0010;
    end
    HOLDP20: begin
      drop = 1'b0;
      credit = 4'b0010;
    end

    CRED3DROP0: begin
      drop = 1'b0;
      credit = 4'b0011;
    end
    HOLDC30: begin
      drop = 1'b0;
      credit = 4'b0011;
    end
    HOLDT30: begin
      drop = 1'b0;
      credit = 4'b0011;
    end
    HOLDP30: begin
      drop = 1'b0;
      credit = 4'b0011;
    end

    CRED0DROP1: begin
      drop = 1'b1;
      credit = 4'b0000;
    end
    HOLDC01: begin
      drop = 1'b0;
      credit = 4'b0000;
    end
    HOLDT01: begin
      drop = 1'b0;
      credit = 4'b0000;
    end
    HOLDP01: begin
      drop = 1'b0;
      credit = 4'b0000;
    end

    CRED1DROP1: begin
      drop = 1'b1;
      credit = 4'b0001;
    end
    HOLDC11: begin
      drop = 1'b0;
      credit = 4'b0001;
    end
    HOLDT11: begin
      drop = 1'b0;
      credit = 4'b0001;
    end
    HOLDP11: begin
      drop = 1'b0;
      credit = 4'b0001;
    end

    CRED2DROP1: begin
      drop = 1'b1;
      credit = 4'b0010;
    end
    HOLDC21: begin
      drop = 1'b0;
      credit = 4'b0010;
    end
    HOLDT21: begin
      drop = 1'b0;
      credit = 4'b0010;
    end
    HOLDP21: begin
```

```
      drop = 1'b0;
      credit = 4'b0010;
    end

    CRED3DROP1: begin
      drop = 1'b1;
      credit = 4'b0011;
    end
    HOLDC31: begin
      drop = 1'b0;
      credit = 4'b0011;
    end
    HOLDT31: begin
      drop = 1'b0;
      credit = 4'b0011;
    end
    HOLDP31: begin
      drop = 1'b0;
      credit = 4'b0011;
    end
  endcase
end

always_ff @(posedge clock, negedge reset_L)
    if (~reset_L)
      currState <= NOCREDIT;
    else
      currState <= nextState;

endmodule: myAbstractFSMExpanded


module ZorgGame // Task 2 Datapath
  (input logic [1:0] CoinValue, ShapeLocation,
  input logic CoinInserted, StartGame, GradeIt, LoadShapeNow, reset, clock,
  input logic [11:0] Guess,
  input logic [2:0] LoadShape,
  output logic [3:0] Znarly, Zood, RoundNumber, NumGames,
  output logic [11:0] masterPattern,
  output logic GameWon, maxRound, doneMasterPattern, syncGradeIt);

  logic numGameEn, numGameClr, numGameUp, notPaid, maxGames, shapeEn1,
        shapeEn2, shapeClr1, shapeClr2, firstLoc,
        secondLoc, thirdLoc, fourthLoc, validLocation, roundEn, roundClr,
        guessedIt, underRoundLimit, syncCoinInserted,
        syncStartGame, syncLoadShapeNow, syncReset,
        drop, nextGame, Gclr, Gload, ldZnarly, ldZood, clrZnarly, clrZood;

  logic [11:0] newShape, moveShapeOut, savedShiftedShapeOut,
               choosePositionOut;
  logic [1:0] checkLocationOut;
  logic [3:0] shiftedByValue, numGamesOut, credit;

  //Lab3 (Input here)
  myAbstractFSMExpanded takeCoins(.credit, .drop, .CoinValue,
                                  .CoinInserted_L(syncCoinInserted),
                                  .clock, .reset_L(syncReset));

  Synchronizer sync1(.async(CoinInserted), .clock, .sync(syncCoinInserted));
  Synchronizer sync2(.async(StartGame), .clock, .sync(syncStartGame));
  Synchronizer sync3(.async(GradeIt), .clock, .sync(syncGradeIt));
  Synchronizer sync4(.async(LoadShapeNow), .clock, .sync(syncLoadShapeNow));
  Synchronizer sync5(.async(reset), .clock, .sync(syncReset));

  masterPatternFSM f1(.StartGame_L(syncStartGame), .GradeIt_L(syncGradeIt),
                  .LoadShapeNow_L(syncLoadShapeNow), .reset_L(syncReset),
                  .GameWon, .*);
  numOfGameFSM f2(.reset_L(syncReset), .*);

  Counter #(4) numberOfGames(.en(numGameEn), .clear(numGameClr),
                             .up(numGameUp), .load(1'b0), .clock, .D(),
                             .Q(numGamesOut));

  assign NumGames = numGamesOut;

  Counter #(4) numberOfRounds(.en(roundEn), .clear(roundClr), .load(1'b0),
                              .up(1'b1), .clock, .D(), .Q(RoundNumber));

  Comparator #(4) paidOrNot(.A(numGamesOut), .B(4'd0), .AeqB(notPaid));
  Comparator #(4) maxNumberOfGames(.A(numGamesOut), .B(4'd7), .AeqB(maxGames));
  Comparator #(4) guess(.A(Znarly), .B(4'd4), .AeqB(guessedIt));

  Comparator #(3) locationOne(.A(masterPattern[2:0]), .B(3'd0),
                              .AeqB(firstLoc));
  Comparator #(3) locationTwo(.A(masterPattern[5:3]), .B(3'd0),
                              .AeqB(secondLoc));
  Comparator #(3) locationThird(.A(masterPattern[8:6]), .B(3'd0),
                              .AeqB(thirdLoc));
  Comparator #(3) locationFourth(.A(masterPattern[11:9]), .B(3'd0),
                              .AeqB(fourthLoc));

  MagComp #(4) c1(.A(RoundNumber), .B(4'd8), .AltB(underRoundLimit), .AgtB(),
               .AeqB(maxRound));

  always_comb begin
    newShape[11:3] = 9'b0;
    newShape[2:0] = LoadShape[2:0];
    shiftedByValue = 4'd0;
    if (ShapeLocation == 2'b00)
      shiftedByValue = 4'd0;
    else if (ShapeLocation == 2'b01)
      shiftedByValue = 4'd3;
    else if (ShapeLocation == 2'b10)
      shiftedByValue = 4'd6;
    else if (ShapeLocation == 2'b11)
      shiftedByValue = 4'd9;
  end

  BarrelShifter moveShape(.V(newShape), .by(shiftedByValue),
                          .S(moveShapeOut));

  Register #(12) saveShiftValue(.en(shapeEn1), .clear(shapeClr1), .clock,
                                .D(moveShapeOut), .Q(savedShiftedShapeOut));
  Register #(12) finalPos(.en(shapeEn2), .clear(shapeClr2), .clock,
                          .D(choosePositionOut), .Q(masterPattern));

  assign choosePositionOut = masterPattern | savedShiftedShapeOut;

  Multiplexer #(4) m1(.I({fourthLoc, thirdLoc, secondLoc, firstLoc}),
                    .S(ShapeLocation), .Y(validLocation));

  assign doneMasterPattern = ~(firstLoc | secondLoc | thirdLoc | fourthLoc);
  assign nextGame = GameWon | maxRound;

  Grader grade(.Guess, .masterPattern, .Gclr, .Gload,
               .clock(clock), .Znarly(Znarly), .Zood(Zood), .doneMasterPattern,
               .ldZnarly, .ldZood, .clrZnarly, .clrZood);

endmodule: ZorgGame


module masterPatternFSM // FSM for controlling game based on masterPattern input
  (input logic StartGame_L, GradeIt_L, LoadShapeNow_L, notPaid,
   doneMasterPattern, validLocation, guessedIt, underRoundLimit, maxRound,
   reset_L, clock,
   output logic GameWon, shapeClr1, shapeClr2, roundEn, roundClr, shapeEn1,
   shapeEn2, Gclr, Gload, ldZnarly, ldZood, clrZnarly, clrZood);

  enum logic [3:0] {INIT, PREP, FINISHPREP, FINISH, EXIT, INCROUND, WON, HOLD1,
                    HOLD2} currState, nextState;

  //Next State Logic
  always_comb begin
    nextState = INIT;
    case(currState)
      INIT: begin
        if (StartGame_L)
          nextState = INIT;
        else if (~StartGame_L && ~notPaid)
          nextState = PREP;
      end
      PREP: begin
        if (LoadShapeNow_L)
          nextState = PREP;
        else if (~LoadShapeNow_L)
          nextState = FINISHPREP;
      end
      FINISHPREP: begin
        if (~validLocation & LoadShapeNow_L)
          nextState = PREP;
        else if (validLocation & LoadShapeNow_L)
          nextState = FINISH;
        else if (~LoadShapeNow_L)
          nextState = FINISHPREP;
      end
      FINISH: begin
        if (~doneMasterPattern)
          nextState = HOLD1;
        else if (doneMasterPattern)
          nextState = EXIT;
      end
      HOLD1: begin
        if (~doneMasterPattern)
          nextState = PREP;
        else if (doneMasterPattern)
          nextState = EXIT;
      end
      EXIT: begin
        if (maxRound)
          nextState = INIT;
        else if (~GradeIt_L & ~guessedIt & ~maxRound)
          nextState = HOLD2;
        else if (GradeIt_L)
          nextState = EXIT;
      end
      HOLD2: begin
        if (~GradeIt_L)
          nextState = HOLD2;
        else if(GradeIt_L)
          nextState = INCROUND;
      end
      INCROUND:
        if (guessedIt & underRoundLimit)
          nextState = WON;
        else
          nextState = EXIT;
      WON: begin
        nextState = INIT;
      end
    endcase
  end

  //Output Logic
  always_comb begin
    shapeClr1 = 0; shapeClr2 = 0; roundClr = 0; roundEn = 0;
    shapeEn1 = 0; shapeEn2 = 0; GameWon = 0; Gclr = 0; Gload = 0; ldZnarly = 0;
    ldZood = 0; clrZnarly = 0; clrZood = 0;
    unique case(currState)
      INIT: begin
        shapeClr1 = 1;
        shapeClr2 = 1;
        roundClr = 1;
        roundEn = 0;
        shapeEn1 = 0;
        shapeEn2 = 0;
        GameWon = 0;
        Gclr = 1;
        Gload = 0;
        ldZnarly = 0;
        ldZood = 0;
        clrZnarly = 1;
        clrZood = 1;
      end
      PREP: begin
        shapeClr1 = 1;
        shapeClr2 = 0;
        roundClr = 0;
        roundEn = 0;
        shapeEn1 = 0;
        shapeEn2 = 0;
        Gclr = 0;
        Gload = 0;
        ldZnarly = 0;
        ldZood = 0;
        clrZnarly = 0;
        clrZood = 0;
      end
      FINISHPREP: begin
        shapeClr1 = 0;
        shapeClr2 = 0;
        roundClr = 0;
        roundEn = 0;
        shapeEn1 = 1;
        shapeEn2 = 0;
        Gclr = 0;
        Gload = 0;
        ldZnarly = 0;
        ldZood = 0;
        clrZnarly = 0;
        clrZood = 0;
      end
      FINISH: begin
        shapeClr1 = 1;
        shapeClr2 = 0;
        roundClr = 0;
        roundEn = 0;
        shapeEn1 = 0;
        shapeEn2 = 1;
        Gclr = 0;
        Gload = 0;
        ldZnarly = 0;
        ldZood = 0;
        clrZnarly = 0;
        clrZood = 0;
      end
      HOLD1: begin
        shapeClr1 = 1;
        shapeClr2 = 0;
        roundClr = 0;
        roundEn = 0;
        shapeEn1 = 0;
        shapeEn2 = 1;
        Gclr = 0;
        Gload = 0;
        ldZnarly = 0;
        ldZood = 0;
        clrZnarly = 0;
        clrZood = 0;
      end
      EXIT: begin
        shapeClr1 = 0;
        shapeClr2 = 0;
        roundClr = 0;
        roundEn = 0;
        shapeEn1 = 0;
        shapeEn2 = 0;
        Gclr = 1;
        Gload = 0;
        ldZnarly = 0;
        ldZood = 0;
        clrZnarly = 0;
        clrZood = 0;
      end
      HOLD2: begin
        shapeClr1 = 0;
        shapeClr2 = 0;
        roundClr = 0;
        roundEn = 0;
        shapeEn1 = 0;
        shapeEn2 = 0;
        Gclr = 0;
        Gload = 1;
        ldZnarly = 1;
        ldZood = 1;
        clrZnarly = 0;
        clrZood = 0;
      end
      INCROUND: begin
        shapeClr1 = 0;
        shapeClr2 = 0;
        roundClr = 0;
        roundEn = 1;
        shapeEn1 = 0;
        shapeEn2 = 0;
        Gclr = 1;
        Gload = 0;
        ldZnarly = 0;
        ldZood = 0;
        clrZnarly = 1;
        clrZood = 1;
      end
      WON: begin
        shapeClr1 = 1;
        shapeClr2 = 1;
        roundClr = 0;
        roundEn = 0;
        shapeEn1 = 0;
        shapeEn2 = 0;
        GameWon = 1;
        Gclr = 1;
        Gload = 0;
        ldZnarly = 0;
        ldZood = 0;
        clrZnarly = 1;
        clrZood = 1;
      end
    endcase
  end

  always_ff @(posedge clock, negedge reset_L)
    if (~reset_L)
      currState <= INIT;
    else
      currState <= nextState;

endmodule: masterPatternFSM


module numOfGameFSM // FSM for controlling number of games
  (input logic drop, maxGames, nextGame, reset_L, clock,
  output logic numGameEn, numGameClr, numGameUp);

  enum logic [1:0] {INIT, PAID, STOP, REMOVEGAME} currState, nextState;

  //Next State Logic
  always_comb begin
    nextState = INIT;
    case(currState)
      INIT: begin
        if (~drop | maxGames)
          nextState = INIT;
        else if (drop & ~maxGames)
          nextState = PAID;
      end
      PAID: begin
        if (drop & ~maxGames & ~nextGame)
          nextState = PAID;
        else if (~nextGame & (~drop | maxGames))
          nextState = STOP;
        else if (nextGame)
          nextState = REMOVEGAME;
      end
      STOP: begin
        if (~nextGame & (~drop | maxGames))
          nextState = STOP;
        else if (drop & ~maxGames & ~nextGame)
          nextState = PAID;
        else if (nextGame)
          nextState = REMOVEGAME;
      end
      REMOVEGAME: begin
        nextState = STOP;
      end
    endcase
  end

  //Output Logic
  always_comb begin
    numGameEn = 0; numGameClr = 0; numGameUp = 0;
    unique case(currState)
      INIT: begin
        numGameEn = 0;
        numGameClr = 1;
        numGameUp = 0;
      end
      PAID: begin
        numGameEn = 1;
        numGameClr = 0;
        numGameUp = 1;
      end
      STOP: begin
        numGameEn = 0;
        numGameClr = 0;
        numGameUp = 1;
      end
      REMOVEGAME: begin
        numGameEn = 1;
        numGameClr = 0;
        numGameUp = 0;
      end
    endcase
  end

  always_ff @(posedge clock, negedge reset_L)
    if (~reset_L)
      currState <= INIT;
    else
      currState <= nextState;

endmodule: numOfGameFSM


// Count Znarlys and Zoods in Guess compared to masterPattern
module Grader
  (input logic [11:0] Guess, masterPattern,
   input logic Gclr, Gload, clock, doneMasterPattern, ldZnarly, ldZood,
   input logic clrZnarly, clrZood,
   output logic AeqB1, AeqB2, AeqB3, AeqB4,
   output logic [11:0] Guess_pos,
   output logic [3:0] Znarly, Zood, sum1, sum2, sum3, sum4, sum5, sum6, sum7);

   logic [3:0] znarlyOut, zoodOut, finalZnarly, finalZood;

   // Znarly Counter //
   Register #(12) r1(.D(Guess), .Q(Guess_pos),
                     .clock(clock), .en(Gload), .clear(Gclr));

   Comparator #(4) znc1(.A(Guess_pos[2:0]), .B(masterPattern[2:0]),
                          .AeqB(AeqB1));
   Comparator #(4) znc2(.A(Guess_pos[5:3]), .B(masterPattern[5:3]),
                          .AeqB(AeqB2));
   Comparator #(4) znc3(.A(Guess_pos[8:6]), .B(masterPattern[8:6]),
                          .AeqB(AeqB3));
   Comparator #(4) znc4(.A(Guess_pos[11:9]), .B(masterPattern[11:9]),
                          .AeqB(AeqB4));

   Adder #(4) zna1(.A(AeqB1), .B(AeqB2), .sum(sum1), .cin(0), .cout());
   Adder #(4) zna2(.A(AeqB3), .B(AeqB4), .sum(sum2), .cin(0), .cout());
   Adder #(4) zna3(.A(sum1), .B(sum2), .sum(znarlyOut), .cin(0), .cout());

   // Zood Counter //
   logic [2:0] Tshape, Cshape, Oshape, Dshape, Ishape, Zshape;
   logic [3:0] T_count, C_count, O_count, D_count, I_count, Z_count;

   assign Tshape = 3'b001;
   assign Cshape = 3'b010;
   assign Oshape = 3'b011;
   assign Dshape = 3'b100;
   assign Ishape = 3'b101;
   assign Zshape = 3'b110;

   // Compare number of specific shape in Guess vs. masterPattern
   Shape_Counter tc(.Shape(Tshape),
                    .partial_Zoods(T_count), .Guess_pos(Guess_pos),
                    .masterPattern(masterPattern));
   Shape_Counter cc(.Shape(Cshape),
                    .partial_Zoods(C_count), .Guess_pos(Guess_pos),
                    .masterPattern(masterPattern));
   Shape_Counter oc(.Shape(Oshape),
                    .partial_Zoods(O_count), .Guess_pos(Guess_pos),
                    .masterPattern(masterPattern));
   Shape_Counter dc(.Shape(Dshape),
                    .partial_Zoods(D_count), .Guess_pos(Guess_pos),
                    .masterPattern(masterPattern));
   Shape_Counter ic(.Shape(Ishape),
                    .partial_Zoods(I_count), .Guess_pos(Guess_pos),
                    .masterPattern(masterPattern));
   Shape_Counter zc(.Shape(Zshape),
                    .partial_Zoods(Z_count), .Guess_pos(Guess_pos),
                    .masterPattern(masterPattern));

   Adder #(3) zoa1(.A(T_count), .B(C_count), .sum(sum3), .cin(0), .cout());
   Adder #(3) zoa2(.A(O_count), .B(D_count), .sum(sum4), .cin(0), .cout());
   Adder #(3) zoa3(.A(I_count), .B(Z_count), .sum(sum5), .cin(0), .cout());
   Adder #(3) zoa4(.A(sum3), .B(sum4), .sum(sum6), .cin(0), .cout());
   Adder #(3) zoa5(.A(sum6), .B(sum5), .sum(sum7), .cin(0), .cout());

   // Subtract Znarly count from shape count to obtain Zood count
   Subtracter #(3) sub(.A(sum7), .B(znarlyOut), .diff(zoodOut), .bin(0),
                       .bout());

   Register #(4) z1(.D(znarlyOut), .Q(finalZnarly),
                     .clock(clock), .en(ldZnarly), .clear(clrZnarly));
   Register #(4) z2(.D(zoodOut), .Q(finalZood),
                     .clock(clock), .en(ldZood), .clear(clrZood));

   always_comb begin
    if (~doneMasterPattern) begin
      Znarly = 4'b0;
      Zood = 4'b0;
    end
    else begin
      Znarly = finalZnarly;
      Zood = finalZood;
    end
   end

endmodule: Grader


// Count number of shapes in Guess and masterPattern
module Shape_Counter
  (input logic [11:0] Guess_pos, masterPattern,
   input logic [2:0] Shape,
   output logic [2:0] partial_Zoods,
   output logic mp_AeqB1, mp_AeqB2, mp_AeqB3, mp_AeqB4,
   output logic [3:0] mp_sum1, mp_sum2, mp_sum3,
   output logic g_AeqB1, g_AeqB2, g_AeqB3, g_AeqB4,
   output logic [3:0] g_sum1, g_sum2, g_sum3,
   output logic mag_AgtB);

  // Count Shape in masterPattern
  Comparator #(3) zocmp1(.A(Shape), .B(masterPattern[2:0]),
                          .AeqB(mp_AeqB1));
  Comparator #(3) zocmp2(.A(Shape), .B(masterPattern[5:3]),
                          .AeqB(mp_AeqB2));
  Comparator #(3) zocmp3(.A(Shape), .B(masterPattern[8:6]),
                          .AeqB(mp_AeqB3));
  Comparator #(3) zocmp4(.A(Shape), .B(masterPattern[11:9]),
                          .AeqB(mp_AeqB4));

  Adder #(3) zoamp1(.A(mp_AeqB1), .B(mp_AeqB2), .sum(mp_sum1),
                    .cin(0), .cout());
  Adder #(3) zoamp2(.A(mp_AeqB3), .B(mp_AeqB4), .sum(mp_sum2),
                    .cin(0), .cout());
  Adder #(3) zoamp3(.A(mp_sum1), .B(mp_sum2), .sum(mp_sum3),
                    .cin(0), .cout());

  // Count Shape in Guess
  Comparator #(3) zocg1(.A(Shape), .B(Guess_pos[2:0]),
                        .AeqB(g_AeqB1));
  Comparator #(3) zocg2(.A(Shape), .B(Guess_pos[5:3]),
                        .AeqB(g_AeqB2));
  Comparator #(3) zocg3(.A(Shape), .B(Guess_pos[8:6]),
                        .AeqB(g_AeqB3));
  Comparator #(3) zocg4(.A(Shape), .B(Guess_pos[11:9]),
                        .AeqB(g_AeqB4));

  Adder #(3) zoag1(.A(g_AeqB1), .B(g_AeqB2), .sum(g_sum1),
                   .cin(0), .cout());
  Adder #(3) zoag2(.A(g_AeqB3), .B(g_AeqB4), .sum(g_sum2),
                   .cin(0), .cout());
  Adder #(3) zoag3(.A(g_sum1), .B(g_sum2), .sum(g_sum3),
                   .cin(0), .cout());

  // Compare masterPattern and Guess Shape counts
  MagComp #(3) mg(.A(mp_sum3), .B(g_sum3),
                  .AltB(), .AeqB(), .AgtB(mag_AgtB));

  // Select lowest count
  Mux2to1 #(3) mult(.I0(mp_sum3), .I1(g_sum3), .S(mag_AgtB), .Y(partial_Zoods));

endmodule: Shape_Counter


//Helps to display the variables we defined onto the FPGA board BCDs
module SevenSegmentDisplay
(input logic [3:0] BCD7, BCD6, BCD5, BCD4, BCD3, BCD2, BCD1, BCD0,
input logic [7:0] blank,
output logic [6:0] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);

logic [6:0] preHEX7, preHEX6, preHEX5, preHEX4, preHEX3, preHEX2, preHEX1,
            preHEX0;
logic [6:0] nonInvertedHEX7,nonInvertedHEX6,nonInvertedHEX5,nonInvertedHEX4,
            nonInvertedHEX3,nonInvertedHEX2, nonInvertedHEX1, nonInvertedHEX0;

BCDtoSevenSegment d0(.bcd(BCD0), .segment(preHEX0));
BCDtoSevenSegment d1(.bcd(BCD1), .segment(preHEX1));
BCDtoSevenSegment d2(.bcd(BCD2), .segment(preHEX2));
BCDtoSevenSegment d3(.bcd(BCD3), .segment(preHEX3));
BCDtoSevenSegment d4(.bcd(BCD4), .segment(preHEX4));
BCDtoSevenSegment d5(.bcd(BCD5), .segment(preHEX5));
BCDtoSevenSegment d6(.bcd(BCD6), .segment(preHEX6));
BCDtoSevenSegment d7(.bcd(BCD7), .segment(preHEX7));

Mux2to1 m0(.I0(preHEX0), .I1(7'b0), .S(blank[0]), .Y(nonInvertedHEX0));
Mux2to1 m1(.I0(preHEX1), .I1(7'b0), .S(blank[1]), .Y(nonInvertedHEX1));
Mux2to1 m2(.I0(preHEX2), .I1(7'b0), .S(blank[2]), .Y(nonInvertedHEX2));
Mux2to1 m3(.I0(preHEX3), .I1(7'b0), .S(blank[3]), .Y(nonInvertedHEX3));
Mux2to1 m4(.I0(preHEX4), .I1(7'b0), .S(blank[4]), .Y(nonInvertedHEX4));
Mux2to1 m5(.I0(preHEX5), .I1(7'b0), .S(blank[5]), .Y(nonInvertedHEX5));
Mux2to1 m6(.I0(preHEX6), .I1(7'b0), .S(blank[6]), .Y(nonInvertedHEX6));
Mux2to1 m7(.I0(preHEX7), .I1(7'b0), .S(blank[7]), .Y(nonInvertedHEX7));

assign HEX0 = ~nonInvertedHEX0;
assign HEX1 = ~nonInvertedHEX1;
assign HEX2 = ~nonInvertedHEX2;
assign HEX3 = ~nonInvertedHEX3;
assign HEX4 = ~nonInvertedHEX4;
assign HEX5 = ~nonInvertedHEX5;
assign HEX6 = ~nonInvertedHEX6;
assign HEX7 = ~nonInvertedHEX7;

endmodule: SevenSegmentDisplay


//Converts the BCDs into the seven segments for the displays on the FPGA
module BCDtoSevenSegment
  (input logic [3:0] bcd,
  output logic [6:0] segment);

  always_comb begin
    unique case(bcd)
      4'b0000: segment = 7'b011_1111;
      4'b0001: segment = 7'b000_0110;
      4'b0010: segment = 7'b101_1011;
      4'b0011: segment = 7'b100_1111;
      4'b0100: segment = 7'b110_0110;
      4'b0101: segment = 7'b110_1101;
      4'b0110: segment = 7'b111_1101;
      4'b0111: segment = 7'b000_0111;
      4'b1000: segment = 7'b111_1111;
      4'b1001: segment = 7'b110_0111;
      default: segment = 7'b000_0000;
    endcase
  end
endmodule: BCDtoSevenSegment


module ZorgGame_test; // Testbench for ZorgGame
  logic [1:0] CoinValue, ShapeLocation;
  logic CoinInserted, StartGame, GradeIt, LoadShapeNow, reset, clock, GameWon;
  logic maxRound;
  logic [11:0] Guess, masterPattern;
  logic [2:0] LoadShape;
  logic [3:0] Znarly, Zood, RoundNumber, NumGames;
  logic doneMasterPattern;

  ZorgGame DUT(.*);

  initial begin
    clock = 0;
    forever #5 clock = ~clock;
  end

  initial begin
    $monitor($time,, "Coin = %b location = %b coinInserted = %b start = %b",
    CoinValue, ShapeLocation, CoinInserted, StartGame,
    " grade = %b loadShape = %b loadNow = %b guess = %b", GradeIt, LoadShape,
    LoadShapeNow, Guess,
    " Znarly = %d Zood = %d, RoundNumber = %d, NumGames = %d Won = %d",
    Znarly, Zood, RoundNumber, NumGames, GameWon);

    //Initial Values
    reset <= 0; CoinValue <= 2'b01; ShapeLocation <= 2'b00; CoinInserted <= 1;
    StartGame <= 1; GradeIt <= 1; LoadShapeNow <= 1; LoadShape <= 3'b110;
    Guess <= 12'b010_001_110_001;

    @(posedge clock);

    reset <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    StartGame <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    StartGame <= 1; LoadShapeNow <= 1; CoinInserted <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinValue <= 2'b10; CoinInserted <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinValue <= 2'b11; CoinInserted <= 0;
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    CoinInserted <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    StartGame <= 0;
    @(posedge clock);
    StartGame <= 1;
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 0; ShapeLocation <= 2'b10;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 0; LoadShape <= 3'b110;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 0; LoadShape <= 3'b001; ShapeLocation <= 2'b00;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 0; LoadShape <= 3'b110;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 0; LoadShape <= 3'b101; ShapeLocation <= 2'b11;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
     LoadShapeNow <= 0; LoadShape <= 3'b100; ShapeLocation <= 2'b01;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    LoadShapeNow <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0; Guess <= 12'b101_011_001_110;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0; Guess <= 12'b101_011_001_110;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0; Guess <= 12'b101_011_001_110;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0; Guess <= 12'b101_011_001_110;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0; Guess <= 12'b101_011_001_110;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 0; Guess <= 12'b101_110_100_001;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    GradeIt <= 1;
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    @(posedge clock);
    #1 $finish;
  end

endmodule: ZorgGame_test
```